Universidad Autónoma de Madrid

Escuela Politécnica Superior

Máster en Ingeniería Informática

TRABAJO DE FIN DE MÁSTER

DESARROLLO DE UN SISTEMA DE ALMACENAMIENTO Y PROCESADO DE LOGS

Paula Roquero Fuentes
Tutor: Dr. Javier Aracil Rico

9 de Septiembre de 2016

DESARROLLO DE UN SISTEMA DE ALMACENAMIENTO Y PROCESADO DE LOGS

Autor: Paula Roquero Fuentes
Tutor: Dr. Javier Aracil Rico

Departamento de Tecnología Electrónica y de las Comunicaciones
Escuela Politécnica Superior
Universidad Autónoma de Madrid

9 de Septiembre de 2016

Abstract

Abstract —

In this document a high performance log storage and processing system is designed and developed. This system can process several million logs per second using a single server and is designed to scale horizontally. The only limit to this scalability is the bandwidth of the links connecting the servers.

Processing logs can provide insights into the state of the devices that generated them. These devices span from high performance servers in datacenters to the small devices that form part of the Internet of Things. The increase in the number of devices makes necessary the development of new systems capable of coping with the higher number of logs.

This work is motivated by the fact that unless tens or hundreds of servers are used, none of the existing systems is capable of processing a high volume of logs. The cause is that these systems have prioritized ease of use and flexibility over performance. The system developed in this work seeks to strike a balance between high performance and flexibility. A distributed lambda architecture has been used to acheive this goal. In the first stage of the lambda architecture the ingested logs are saved to persistent storage. The second stage runs in parallel, taking a sample of the received logs for real time processing and analysis. The sampled logs can be forwarded to one of the existing systems or processed with custom software capable of detecting problems or extracting analytics. When a problem is detected the persistent storage can be accessed and the recovered logs are analyzed to get more information.

Key words — Log, Database, High Performance, Big Data, Lambda Architecture

Resumen

Resumen —

En este trabajo de fin de máster se diseña y desarrolla un sistema de alto rendimiento para el almacenamiento y procesamiento de logs. El sistema es capaz de tratar con varios millones de logs por segundo usando un único servidor y su diseño permite hacer un escalado horizontal limitado únicamente por el ancho de banda de las redes que conectan los servidores.

El procesado de logs resulta importante debido a que permiten conocer el estado de los dispositivos que los generan. Estos pueden ir desde pequeños dispositivos que forman parte del Internet of Things hasta servidores ejecutándose en equipos de alto rendimiento. Debido al aumento de el número de estos dispositivos es necesario el uso de sistemas cada vez más potentes para tratar con la cantidad de logs que generan.

La motivación de este trabajo viene dada porque ninguna de las soluciones existentes es capaz de procesar un gran volumen de logs sin usar decenas o centenas de servidores, ya que se centran en tener mayor flexibilidad y facilidad de uso. Por otro lado, el sistema desarrollado en este trabajo permite conseguir alto rendimiento y flexibilidad gracias al uso de una arquitectura lambda distribuida. Todos los logs que llegan al sistema son almacenados de forma persistente al mismo tiempo que se toma y procesa una pequeña fracción de los logs para su análisis en tiempo real. Estos logs pueden ser introducidos en alguna de las soluciones ya existentes o procesados con software diseñado a medida para detectar incidencias u obtener analíticas. En caso de detectar una incidencia se podrá recurrir al almacenamiento persistente para leer todos los logs y obtener una visión completa del problema.

Palabras clave — Log, Base de datos, Alto rendimiento, Big Data, Arquitectura Lambda

Acrónimos

HDFS Hadoop Distributed File System. 2

IoT Internet of Things. 1

Índice general

Índice de tablas

Índice de figuras

1

Introducción

En los últimos años se ha dado un aumento del número de dispositivos conectados a internet. Estos dispositivos van desde servidores encargados de servir miles de peticiones por segundo hasta dispositivos de bajo consumo que forman parte del llamado Internet of Things.

Estos dispositivos tienen en común la generación de logs o registros que aportan información sobre su estado.

Esta información es muy heterogénea y depende del dispositivo que la genere. En el caso de los servidores estos logs pueden indicar accesos por parte de un cliente o avisar de condiciones de error como un fallo en el servidor. Un ejemplo de este tipo de log se muestra a continuación:

<15>Feb 2 13:13:45 debian apache2: 127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] "GET /apache$_p b.gif HTTP$/1,0" 2002326

Por otro lado, los registros generados por dispositivos Internet of Things (IoT) generalmente son datos recogidos por sensores y consisten en el timestamp y un número que representa la medición.

En cualquiera de estos casos, los logs mantienen un registro del estado de los dispositivos que los generan y, dado que generalmente estos dispositivos no están aislados sino que forman parte de sistemas más complejos, permiten obtener una visión global del estado del sistema.

Es precisamente esta complejidad la que presenta problemas a la hora de monitorizar y diagnosticar los problemas. En un sistema con un elevado número de dispositivos es difícil encontrar la causa de un problema inspeccionando los logs de forma individual. Por esta razón resulta útil tener todos los logs almacenados en un lugar centralizado, de modo que se puedan hacer análisis de datos agregados.

La experiencia previa del laboratorio de investigación en el que se ha realizado este trabajo ha demostrado que, cuando el volumen de logs es muy alto, es necesario dividir el análisis en dos fases bien diferenciadas:

1. La primera fase consiste en un análisis de alto nivel de los datos para detectar incidencias. Este análisis puede realizarse agregando datos obtenidos de los logs para ver posibles variaciones. Gracias a este tipo de análisis se reduce el coste de detectar incidencias, dado que únicamente es necesario observar una pequeña fracción de los datos.

2. La segunda fase sólo entra en acción cuando se detecta una incidencia, ya sea a partir del análisis de los logs a alto nivel o por otros medios. Es entonces cuando resulta de interés acceder a los datos en bruto y ver exactamente cuál ha sido el problema. Una característica de estos datos en bruto es que son volátiles. Su utilidad disminuye según avanza el tiempo y no se detecta una incidencia relacionada con ellos, por lo que pueden ser borrados tras unos pocos días.

Los sistemas de almacenamiento y procesado de logs tienen en cuenta esta división y utilizan arquitecturas lambda. Estas arquitecturas se dividen en dos fases ejecutadas en paralelo. La primera fase almacena los logs en sistemas como Hadoop Distributed File System (HDFS), mientras que la segunda realiza un procesado de los logs usando sistemas como **Storm** o **Spark Streaming**.

Este trabajo busca desarrollar un sistema que, siguiendo este esquema de almacenamiento y procesado, sea capaz de tratar con cantidades masivas de datos que los sistemas actuales no son capaces de procesar.

1.1. Motivación

El almacenamiento y procesamiento de logs es una tarea en la que el laboratorio de investigación donde se ha realizado este trabajo tiene experiencia. Trabajar con logs en entornos de producción ha permitido determinar las limitaciones de los sistemas dedicados al procesamiento de logs como se verá en el estado del arte. Esta experiencia ha mostrado que el volumen de logs con el que hay que tratar empieza a saturar los sistemas actuales. En algunos casos no se pueden guardar todos los logs que se desean y hay que tomar una muestra, perdiendo información. También se espera que en el futuro el volumen de logs aumente aún más.

Frente a este problema se han explorado distintas posibilidades. En primer lugar se han valorado otros sistemas dedicados al procesamiento y almacenamiento de logs para comprobar si es posible utilizar una solución que no necesite desarrollo adicional. Las pruebas han demostrado que, aunque estas soluciones podrían aliviar los problemas actuales, no permitirán procesar el volumen de logs que se espera en el futuro.

Otra posible solución ha sido utilizar algún software de *big data* como **Hadoop** o **Storm** para crear un sistema capaz de procesar logs a partir de herramientas que ya existen. Los resultados han mostrado que, aunque estos sistemas podrían conseguir un rendimiento que se aproxima al que se quiere, su coste sería prohibitivo, ya que es necesario utilizar un gran número de servidores para que estos sistemas escalen horizontalmente.

Esta situación ha llevado a la decisión de crear un nuevo sistema de procesamiento de logs capaz de tratar con el volumen de datos que se desea procesar en el futuro. El sistema debe exprimir al máximo el rendimiento del hardware donde se ejecute, permitiendo al mismo tiempo ser escalado horizontalmente.

1.2. Objetivos

El objetivo de este trabajo es la creación de un sistema capaz de recolectar, almacenar y procesar varios millones de logs por segundo, superando al menos en un orden de magnitud a todas las soluciones existentes.

El diseño del sistema seguirá una arquitectura lambda, con una componente de procesado en tiempo real y otra de almacenamiento centralizado. Esto permitirá usar el sistema para hacer un análisis en dos fases que sigue el esquema explicado anteriormente.

Dado que la mayor prioridad del sistema es conseguir un gran ancho de banda, será aceptable sacrificar otras métricas como la latencia o la fiabilidad. Aunque este segundo punto parece ser un sacrificio demasiado grande, la experiencia en el laboratorio ha demostrado que perder un pequeño porcentaje de los logs no afecta a la capacidad para realizar análisis.

Dada la naturaleza volátil de los logs, la componente de almacenamiento tendrá que guardar únicamente los de la última semana, ayudando así a reducir el coste que puede suponer almacenar todos los logs.

Por otro lado, la componente de proceso se encargará de extraer métricas agregadas y logs especialmente relevantes como por ejemplo los que indiquen condiciones de error. Estas métricas serán introducidas en sistemas de visualización que permitan detectar anomalías visualmente. En este trabajo se busca diseñar un sistema que permita a otros desarrolladores escribir el código que extraiga las métricas. La componente de visualización no forma parte del desarrollo que se llevará a cabo en este trabajo.

1.3. Estructura del documento

En lo que sigue del documento se realizará primero un estudio del estado del arte para comprobar si es posible aprovechar alguna tecnología existente y aprender técnicas que puedan ser útiles a la hora de diseñar el sistema. Después se enumeran los requisitos que debe cumplir el sistema y el diseño que se ha obtenido a partir de los mismos. A continuación se explicará cómo se ha implementado el diseño y las pruebas que se han realizado. Por último, se sacarán conclusiones del trabajo y se explorará el trabajo futuro que servirá para mejorar el sistema.

2

Estado del Arte

La mayoría de sistemas de *big data* existentes están optimizados para escenarios de análisis distintos al que se busca en este trabajo. Su objetivo principal es el de realizar análisis complejos sobre grandes cantidades de datos persistentes que se insertan a poca velocidad. Estos sistemas también optan por ser generales en vez de centrase en un caso de uso más específico.

Los sistemas de *big data* tradicionales pueden escalar horizontalmente para sacar partido a un gran número de nodos. Esto se consigue gracias a algoritmos complejos que dividen la carga de forma transparente al programador, permitiendo enfocarse en el desarrollo de los algoritmos necesarios para solucionar un problema en vez de detalles de más bajo nivel. Sin embargo, esta decisión hace que se sacrifique la escalabilidad vertical de los sistemas, por lo que es necesario usar muchos nodos para ejecutarlos y se incrementa demasiado el coste de operación. Otra decisión de diseño que busca facilitar el trabajo de los programadores es usar lenguajes de alto nivel como Java o Scala para implementar estos sistemas. Sin embargo, esto también hace que sufra la escalabilidad vertical ya que no se consigue la misma velocidad que usando código nativo.

En el diseño de este trabajo la pérdida de escalabilidad vertical no es aceptable. El ancho de banda de datos que debe recibir el sistema de almacenamiento y procesado de logs impone restricciones difíciles de cumplir con sistemas de *big data* tradicionales a menos que se use un gran número de nodos. El aumento de escalabilidad vertical también ayuda a conseguir de forma económica una replicación activo-activo, ya que no es necesario tener tantos servidores ejecutando el sistema. Debido a estas limitaciones se ha decidido diseñar un sistema que permite almacenar y procesar en tiempo real estos datos. Posteriormente, una fracción de los datos puede ser insertada en otros sistemas de más alto nivel como **Elasticsearch** para realizar análisis de forma más sencilla.

En este estado del arte se explorarán las soluciones existentes para guardar y procesar logs con el fin de demostrar que ninguna de ellas cumple los requisitos propuestos en este trabajo. La tabla 2.1 muestra un resumen del rendimiento de las soluciones existentes en el estado del arte para cada parte del sistema.

[1] Transacciones por segundo

2.1. Centralización y distribución de mensajes

La elección más obvia para centralizar los logs sería uno de los demonios de syslog existentes. De todos ellos, **Syslog-ng** es el más rápido al haber sido diseñado con este fin en mente. Puede ser configurado de forma sencilla para recibir logs de demonios de syslog ejecutándose en los dispositivos que generan los logs, procediendo después a guardarlos en una localización centralizada. Esto haría que fuera una elección perfecta para el problema que se plantea pero desgraciadamente, este sistema no es capaz de alcanzar el ancho de banda que se desea, quedándose en 650.000 como máximo [1].

Otro sistema que podría ajustarse con facilidad a la centralización de logs es **Apache Kafka**, un *broker* de mensajes capaz de recibir de diversas fuentes de datos y distribuirlos entre varios destinos. **Kafka** usa un modelo de suscriptor-consumidor que permite guardar los mensajes en almacenamiento persistente al mismo tiempo que se envían a los consumidores que se han suscrito. Estos consumidores pueden ser sistemas de procesado de logs menos eficientes como **Logstash** o **Fluentd**. **Kafka** categoriza los mensajes en *topics*, permitiendo que los consumidores se suscriban únicamente a unos pocos para tener que procesar menos datos. Cada *topic* se divide en varias particiones que pueden ser accedidas en paralelo por múltiples consumidores. Por otro lado, los productores son capaces de publicar sus mensajes a un *topic* específico.

En *benchmarks* publicados por Linkedin [8], un *cluster* formado por tres nodos es capaz de recibir hasta 2 millones de mensajes por segundo generados por tres productores distintos. Estas pruebas también muestran que la tasa de inserción no se ve perjudicada al añadir consumidores, ya que un solo productor es capaz de insertar 800K mensajes por segundo con o sin consumidores.

A diferencia de **Syslog-ng**, **Kafka** no está diseñado específicamente para su uso en el procesamiento de logs, por lo que sería necesario realizar trabajo adicional para ajustarlo a este problema. Dado que presentaba resultados prometedores, se hicieron pruebas con **Kafka** en un servidor con dos `Intel Xeon E5-2630 v2 @ 2.60 GHz`, 32 GB de memoria RAM y un RAID 0 compuesto de 10 discos mecánicos. Las pruebas mostraron que **Kafka** no cumplía con los requisitos propuestos, ya que solo era capaz de consumir 300.000 logs de tamaño medio por segundo usando un productor y un consumidor. Estos resultados son consistentes con los

Tabla 2.1: Resumen de rendimiento

Funcionalidad	Solución	Rendimiento
Centralización y Distribución	Syslog-ng	650K log/s multihilo [1]
	Apache Kafka	300K log/s Usando un productor y un consumidor
	Logstash	29.5K log/s/hilo
	Apache Flume	0.77M log/s/hilo
Base de Datos	Cassandra	251K TPS[1] en un solo nodo [15]
	ScyllaDB	1.8M TPS en un solo nodo [15]
Procesado (Partir y concatenar los campos del log)	Apache Storm	900K logs/s Usando 100 % de todos los cores
	Python	500K log/s/hilo
	AWK	900K log/s/hilo
	Perl / Ruby	300K log/s/hilo

obtenidos por otros sistemas de recolección de logs que usan **Kafka** como base [2]. Este ancho de banda está al menos un orden de magnitud por debajo de lo que se desea, por lo que la posibilidad de usarlo como base del sistema fue descartada.

Una alternativa a **Kafka** es **Apache Flume**. Este sistema es similar a **Kafka**, pero envía directamente los datos a sus destinos en vez de esperar a que un cliente se registre como en **Kafka**. Los datos son recibidos por un *source* que se puede configurar para que sea capaz de interpretar el formato de entrada. Los datos interpretados son guardados en un canal de donde un *sink* los lee para transformarlos al formato de salida deseado y reenviarlos. Existen *sources* y *sinks* capaces de recibir y generar distintos formatos de datos, así como distintas implementaciones de *canales*.

Para probar el rendimiento de **Flume** se usaron dos configuraciones distintas para los *sinks*. Por un lado se usó el *Null Sink*, que descarta todos los eventos que le llegan de un canal. Por otro lado se evaluó el rendimiento del *File Roll Sink*, que guarda los eventos en una serie de ficheros. En este segundo caso se configuró el *sink* para que los ficheros se guardaran a un disco RAM con el fin de conseguir el mayor rendimiento de escritura posible. En el caso del *Null Sink* **Flume** fue capaz de alcanzar un ancho de banda de 1,5 millones de logs por segundo e hilo, mientras que al usar el *File Roll Sink* el rendimiento se quedó en 1,1 millones de logs por segundo e hilo.

Hay otros sistemas de procesamiento que, aunque no dan la tasa, merece la pena comentar. El más conocido es **Logstash**, desarrollado por la empresa *Elastic*. Este sistema recolecta, procesa y reenvía logs usando distintos protocolos de entrada y salida como TCP, UDP, **Elasticsearch**, **Kafka**, **0mq**, etc. Este sistema es capaz de procesar hasta 50K eventos por segundo.

Otro sistema desarrollado como alternativa a **Logstash** es **FluentD**. Aunque los creadores indican que puede llegar a procesar hasta 800K eventos por segundo [18], no especifican detalles de cómo se puede reproducir este resultado [19]. Por otro lado, en su página web indican que el sistema es capaz de procesar 13K eventos por segundo y core [5].

2.2. Base de datos

La primera componente de una arquitectura lambda es el almacenamiento persistente de los datos. Esto permite analizarlos en detalle tras detectar alguna incidencia. Debido al gran volumen de datos que se espera es necesario usar alguna base de datos capaz de funcionar en varios nodos y usar varios discos.

La opción más sencilla sería usar una base de datos distribuida como **Apache HIVE** o **Cassandra** que se encargue de hacer la distribución automáticamente. Sin embargo, las pruebas realizadas con estas tecnologías han mostrado que son necesarios muchos servidores, por lo que estos sistemas fueron descartados [10]. Este problema ha sido abordado previamente por otras soluciones que buscan mantener compatibilidad con estas bases de datos al mismo tiempo que mejoran su rendimiento.

Un ejemplo es **MapRDB**, una distribución de **Hadoop** capaz de insertar 100 millones de puntos por segundo usando 4 nodos y un factor de replicación de 3. Para conseguir esto usan su propio sistema de ficheros llamado **mapR-FS** [13]. Aunque esta base de datos cumple con los requisitos necesarios para el sistema de almacenamiento de logs, su coste de 4000$ por nodo [2] descarta su uso.

[2] https://www.loggly.com/

También hay una base de datos llamada **ScyllaDB** que busca mantener compatibilidad con **Cassandra**. En sus pruebas [16] consiguen 1,9 millones de transacciones por segundo en un cluster de 3 nodos con factor de replicación 3. Cada uno de estos nodos cuenta con 128 GB de memoria RAM y 4 SSDs de 960 GB en un RAID. En otra de sus pruebas [15] consiguen 1,8 millones de transacciones por segundo usando un único nodo. **ScyllaDB** consigue este rendimiento gracias a dos factores distintos. Por un lado, está escrita en C++, consiguiendo de ese modo una aceleración respecto al código Java en el que está escrito **Cassandra**. El segundo factor es el uso de una pila TCP de alto rendimiento que se ejecuta en espacio de usuario gracias a **Intel DPDK**. Aunque estos resultados resultan prometedores, hay que tener en cuenta que han sido realizados en máquinas con RAIDs de SSDs. El gran volumen de logs que se debe guardar hace que esta solución tenga un coste prohibitivo, ya que habría que comprar suficientes discos como para guardar varios TB de datos al día. Este volumen de datos hace necesario el uso de discos mecánicos.

Otra opción que se podría plantear para el almacenamiento es programar una componente para que distribuya los logs a varios nodos donde se guarden en bases de datos clave-valor. Sin embargo, los benchmarks de este tipo de soluciones muestran que no son capaces de almacenar el volumen de datos deseado [3]. Es por ello que se ha decidido que el mejor formato de almacenamiento son ficheros planos donde se escribe de forma secuencial.

2.3. Procesado

La segunda componente de la arquitectura lambda es el sistema de procesado en streaming. A pesar de no tener que procesar todos los datos, es deseable usar una solución que sea capaz de procesar la mayor proporción de logs posible.

El sistema de streaming más usado es **Apache Storm**. Se ejecutó un benchmark de este sistema usando un nodo de doble socket con dos Intel Xeon E5-2630 v2. Usando todos los cores al máximo, **Storm** fue capaz de procesar hasta 900.000 lineas de log por segundo. El proceso consistió en partir los logs por espacios y volver a concatenarlos después. También se probaron **Apache Spark streaming** y **Apache Flink** pero en ambos casos se obtuvo peor rendimiento.

A pesar de que este rendimiento puede parecer muy bueno a primera vista, el mismo proceso y resultados se pueden obtener usando el lenguaje AWK en un único core, frente a los 12 que necesita **Storm**. Otros lenguajes más conocidos como Python son capaces de procesar hasta 500.000 logs por segundo.

Tras observar este resultado se decidió que la mejor solución es usar tuberías de Unix para mandar los logs a procesos externos de AWK o Python que serán los encargados de hacer el proceso. Esta solución tiene como beneficio añadido el hecho de que estos lenguajes son conocidos y fáciles de usar para procesar texto.

2.4. Resumen

En este estudio del estado del arte se ha comprobado que ninguna de las soluciones existentes cumplen con los objetivos que se proponen en este trabajo. Como se ve en la tabla 2.1, los sistemas dedicados al procesado de logs quedan muy lejos del rendimiento que se busca, ya que se centran más en la facilidad de uso. Tampoco es posible crear un sistema juntando varios componentes

existentes ya que, aunque ofrecen más rendimiento, siguen sin cumplir con los requisitos.

Este estudio del estado del arte también ha permitido comprobar que el mayor rendimiento se consigue cuando se usan soluciones escritas en lenguajes compilados como C o C++ o lenguajes especializados como AWK. Estas lecciones son aplicadas en el diseño e implementación que se explican en los siguientes capítulos.

3 Diseño

En esta sección se enumerarán los requisitos detallados que se deben cumplir al diseñar el sistema. Después se explicará cómo evolucionó el diseño y cuál fue el diseño final que se terminó implementando.

3.1. Requisitos

Los requisitos que se listan a continuación han sido derivados de las lecciones aprendidas en el laboratorio de investigación durante la realización de proyectos previos donde se trabajaba con logs. Se busca suplir limitaciones que ya se han encontrado y que el sistema pueda seguir siendo usado en el futuro.

Recibir logs en formato de Syslog.

En primer lugar se desea que el sistema mantenga compatibilidad con los formatos de log existentes. Dado que el formato **Syslog** es el más común, ha sido elegido como método de entrada al sistema. Otra razón para usar este formato estándar es que no es necesario instalar ningún software en el equipo que genere los logs, ya que **Syslog** suele estar instalado en servidores. Este punto es importante, dado que la experiencia del grupo de investigación muestra que los administradores de sistemas generalmente son reacios a instalar nuevo software en sus servidores, sobre todo si están ejecutando un servicio crítico. Esto se debe a que es más probable que un nuevo software tenga algún bug que afecte a la estabilidad o rendimiento del servidor.

Centralizar los logs.

En segundo lugar, se desea recibir todos los logs en un solo punto. Como se explicó en la introducción, esto se hace para facilitar el análisis de los logs, ya que es más fácil acceder a ellos si están centralizados.

Marcar el tiempo de recepción de los logs.

Al recibir logs de diversas fuentes, cada uno tendrá un formato ligeramente distinto. Esto supone un problema a la hora de determinar en qué momento se generó el log, ya que traducir las fechas a un formato uniforme resulta caro computacionalmente. Es por esto que se debe añadir a cada log una marca de tiempo que permita buscarlos más tarde. Aunque esta marca no coincidirá con el tiempo en el que se generó el log en el servidor, permitirá hacer una búsqueda aproximada de los logs de interés para una incidencia. Esto no es un problema dado que generalmente no se conoce el tiempo exacto en el que ocurrió el problema, solo un rango de tiempos. Además, añadir la marca de tiempo en un punto único evita problemas causados por servidores con el tiempo mal sincronizado.

Clasificar los logs.

Los logs deben clasificarse por categorías para permitir que los usuarios del sistema identifiquen logs de varios servicios distintos. Este requisito también permite agilizar las búsquedas en los datos guardados persistentemente, ya que la categoría se puede usar como clave. Como ejemplo, se podrían tener categorías distintas para los logs generados por nodos de servidor web y para los generados por nodos de base de datos.

Almacenar los logs en bruto.

Los logs se deben almacenar en bruto para poder acceder a ellos en un futuro si es necesario obtener información adicional a la conseguida en el análisis de tiempo real. Esto puede ocurrir si se detecta una incidencia. Este requisito se corresponde con la parte de almacenamiento en una arquitectura lambda.

Procesar parte de los logs en tiempo real.

Una parte de los logs debe ser procesada en tiempo real para detectar anomalías o hacer transformaciones sobre los datos que permitan almacenarlos en una base de datos de más alto nivel. Es por ello que se debe proveer algún lenguaje de programación para poder hacer transformaciones y análisis sobre los datos. Este requisito se corresponde con la parte de proceso de la arquitectura lambda.

Guardado de datos durante al menos una semana.

Los datos almacenados pueden ser usados para hacer un análisis en detalle cuando se detecta una incidencia. Sin embargo, la utilidad de estos datos decrece con el tiempo, ya que las incidencias suelen detectarse poco después de ocurrir. Por ello se ha decidido que los logs deben ser guardados durante al menos una semana, considerando que después de ese tiempo pueden ser borrados sin perder información.

API para hacer lecturas de logs.

Se debe crear una API para permitir la lectura de los logs almacenados en disco. La API permitirá filtrar los logs por fechas y categorías. Esta API está pensada para hacer lecturas de logs en un rango de tiempos, no de forma individual.

Envío de logs procesados por red.

Una vez procesados, los logs deben enviarse por red a un destino configurable donde se almacenarán en una base de datos de alto nivel. Esto permite hacer análisis y visualización rápidos de una fracción de los logs.

Velocidad de 3 millones de logs por segundo.

Para tener un objetivo bien definido se ha determinado que el sistema debe ser capaz de procesar y almacenar al menos 3 millones de logs de tamaño medio por segundo. Tras hacer un análisis de logs obtenidos de varios sistemas en producción se determinó que el tamaño típico de los logs se encuentra entre los 100 y 200 bytes como se muestra en la figura 3.1.

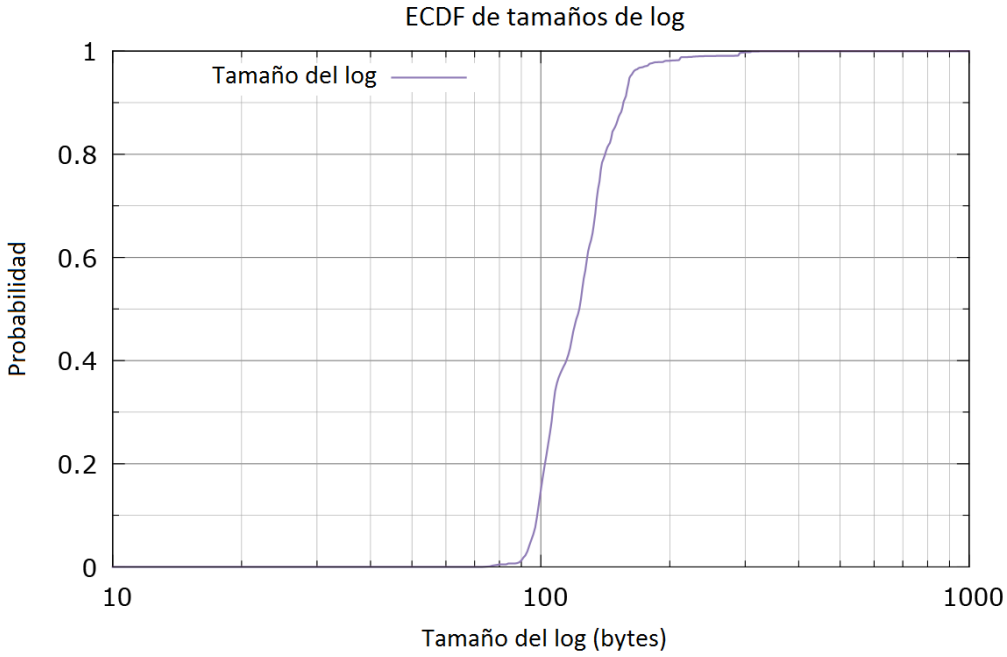

Figura 3.1: ECDF de tamaños de log

Prioridad de escritura.

A la hora de almacenar los datos es vital que el sistema no se bloquee, ya que de lo contrario se podría perder una proporción alta de los logs. Es por ello que se debe dar prioridad a la velocidad de escritura frente a la de lectura. Es aceptable que las lecturas tengan latencias altas.

3.2. Sistema diseñado

En esta sección se explicará el diseño del sistema y las decisiones que se tomaron para llegar a él. Desde el inicio se consideró que el sistema debía estar dividido en tres componentes bien definidas: el centralizador de datos, el sistema de almacenamiento y el sistema de procesado. Esta separación permite cierta flexibilidad a la hora de diseñar el sistema completo y comunicar cada

componente entre sí. La imagen 3.2 muestra un diagrama del sistema completo. Cada componente será explicada en detalle a continuación. Como se explicó en la introducción, la parte del sistema correspondiente a la base de datos de alto nivel no entra dentro del ámbito de este trabajo.

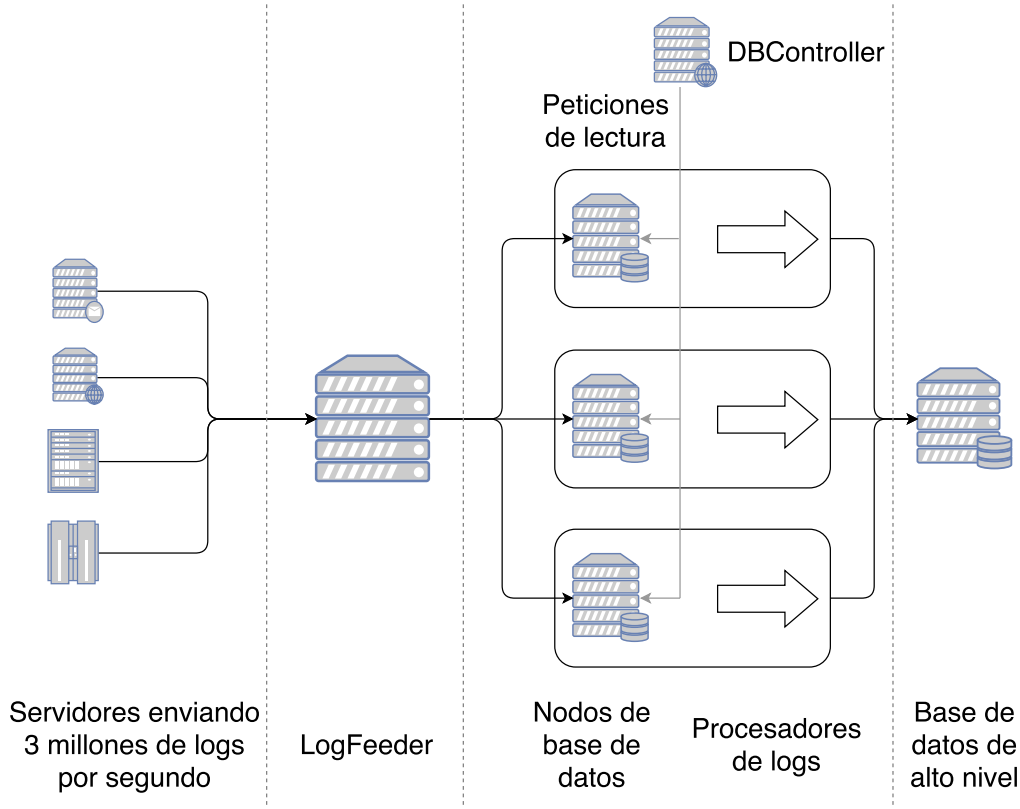

Figura 3.2: Diagrama del sistema completo

3.2.1. LogFeeder

El centralizador (llamado **LogFeeder**) es el punto de entrada de los logs al sistema, por lo que debe ser capaz de procesar los 3 millones de logs por segundo que se han propuesto como requisito. La primera decisión que se tomó fue el uso de UDP como protocolo de transporte para los logs. De este modo es más fácil alcanzar el ancho de banda deseado, ya que tiene menos sobrecoste que otros protocolos como TCP. Además, este protocolo es soportado por defecto por los demonios de **Syslog**, por lo que no es necesario instalar software especializado en los equipos generadores de logs, simplemente hay que cambiar configuraciones. Aunque UDP tiene la desventaja de no ser un sistema fiable, se ha considerado que es aceptable perder una pequeña proporción de logs si de ese modo se consigue alcanzar el ancho de banda deseado. El uso de UDP también permite recibir los datagramas en software que se ejecuta en espacio de usuario sin necesidad de que pasen por la pila de red del sistema. Esto aumenta el ancho de banda y disminuye la latencia, mejorando aún más el rendimiento.

Otra decisión que se tomó inicialmente fue que los timestamp se añadirían a los logs en este punto. De este modo se consigue una medida consistente, ya que los servidores de almacenamiento pueden tener distintas horas configuradas y el sistema puede introducir latencias distintas para cada log, haciendo que se altere el orden de los timestamps.

El centralizador también marca a qué categoría pertenece un log. Estas categorías pueden indicar por ejemplo que el nodo emisor es un servidor HTTP encargado de servir peticiones estáticas. Esta clasificación se basa en dos datos: la IP de origen del log y el nivel de alerta que viene incluido en las cabeceras de syslog. El uso del nivel de alerta permite clasificar los logs sin tener que inspeccionarlos a fondo, operación que gastaría muchos recursos debido a la irregularidad de los logs. De este modo el administrador de sistemas puede poner el mismo código de alerta a todos los logs de un servidor y después asignar en la configuración del sistema cada par IP, nivel de alerta a una categoría determinada.

Al inicio del proyecto el sistema no estaba concebido como una arquitectura lambda, sino que se pensaba hacer que el centralizador filtrara y extrajera los campos útiles de cada log, almacenando los datos resultantes en el disco local. Este diseño terminó siendo descartado, ya que se decidió que era necesario guardar todos los logs en almacenamiento persistente para analizar en profundidad posibles incidencias. Dado que descartar algunos campos de cada log no habría conseguido una disminución sustancial en la cantidad de datos a almacenar, fue necesario modificar el diseño del sistema y usar almacenamiento distribuido en varios equipos. Esto permitió distribuir también la computación, consiguiendo una mejora del rendimiento del sistema al parelizar el procesado de los logs. De este modo se llegó a un diseño similar a una arquitectura lambda, con una componente de almacenamiento persistente y otra de procesado en tiempo real.

La última decisión fue determinar cómo se reenviarían los logs a los nodos de almacenamiento. Se barajó la posibilidad de dividirlos basándose en las categorías, pero este diseño fue descartado debido a la dificultad de balancear correctamente los logs. En su lugar se decidió utilizar una estrategia *round robin* para mandar los logs a los nodos de almacenamiento conectados. Esto evita cuellos de botella si hay un aumento repentino en el número de logs perteneciente a una categoría determinada.

En la figura 3.3 se muestran los distintos procesos que se aplican a un log desde que entra a **LogFeeder** hasta que sale hacia la base de datos.

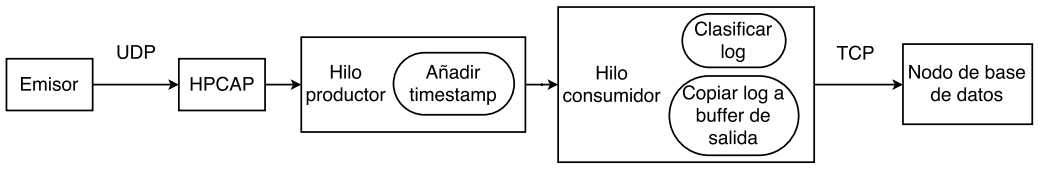

Figura 3.3: Proceso de un log

3.2.2. Nodos de base de datos

Una vez decidido que los logs se almacenarían de forma distribuida, fue necesario determinar cómo se conseguiría el mayor rendimiento. Al estudiar el estado del arte se comprobó que las bases de datos distribuidas perdían más rendimiento cuanto más interdependientes fueran los nodos entre sí. Es por ello que se decidió que cada nodo actuaría de forma independiente, sin conocer la existencia de otros nodos.

Usando este diseño **LogFeeder** es el encargado de dividir los logs entre los nodos de la base de datos de forma uniforme. Como se explicó en la sección anterior, esto se consigue usando un algoritmo *round robin*. También hace falta un punto central al que dirigir las peticiones de lectura que se encargue de redirigir estas peticiones a cada nodo individual de la base de datos.

Este punto central ha sido llamado **DBController**.

Escritura

La componente de escritura es muy simple. Recibe los datos enviados por **LogFeeder** y los almacena de forma asíncrona en ficheros de 1 GB. Este tamaño de fichero ha sido elegido con el fin de aprovechar el rendimiento que ofrecen los discos mecánicos al hacer escrituras secuenciales. También permite obtener el rendimiento máximo de RAIDs 0 de discos mecánicos.

Los nombres de los ficheros están formados por los timestamps del primer y último log. Estos nombres se guardan en una base de datos SQL donde se pueden hacer búsquedas a partir de los timestamps. Esto permite encontrar los ficheros relevantes sin perturbar las operaciones de escritura y lectura, ya que la base de datos SQL puede guardarse en un disco distinto.

Lectura

Las lecturas de la base de datos se dividen en dos fases. En primer lugar es necesario que un cliente haga una petición de lectura a **DBController**. En esta petición se indican los timestamps de inicio y fin de la lectura además de las categorías de los logs a leer y la IP destino a la que se enviarán los resultados del proceso. **DBController** es capaz de encolar varias peticiones de lectura y también puede recibir órdenes para borrar peticiones en cola o parar la lectura que se encuentra en ejecución.

En la segunda fase **DBController** inicia una lectura, enviando una petición a cada nodo individual de la base de datos. Tras recibir la petición, cada nodo individual busca los ficheros que debe leer en la base de datos SQL y los lee de forma secuencial. Cada fichero se lee entero y se almacena en un buffer con el fin de disminuir el número de operaciones de entrada/salida. Una vez se ha leído el fichero completo, se realiza el procesado pertinente sobre los logs pertenecientes a las categorías especificadas en la petición de lectura y se mandan los resultados por red a la base de datos de alto nivel. Cada nodo individual también puede recibir peticiones de cancelación de lectura enviados por **DBController**.

La figura 3.4 muestra este proceso de lectura.

3.2.3. Procesado

El sistema de procesado tiene como objetivo filtrar y transformar los logs. De este modo el gran volumen de datos que recibe el sistema se reduce lo suficiente como para enviar una muestra a una base de datos de más alto nivel como las estudiadas en el estado del arte. Usando estas bases de datos de alto nivel es posible obtener analíticas en tiempo real que permiten detectar incidencias rápidamente. Después se puede acceder a los datos completos guardados en la base de datos y hacer un análisis completo. A estos datos también se les puede aplicar un procesado antes de ser enviados a la base de datos de alto nivel.

Cuando se cambió el diseño del sistema para convertirlo en un sistema distribuido, se decidió mover la componente de filtrado y procesado a los nodos de almacenamiento con el fin de aprovechar los recursos de computación de todas las máquinas. De este modo el procesado sigue una estructura map-reduce en la que **LogFeeder** divide los logs entre los nodos de

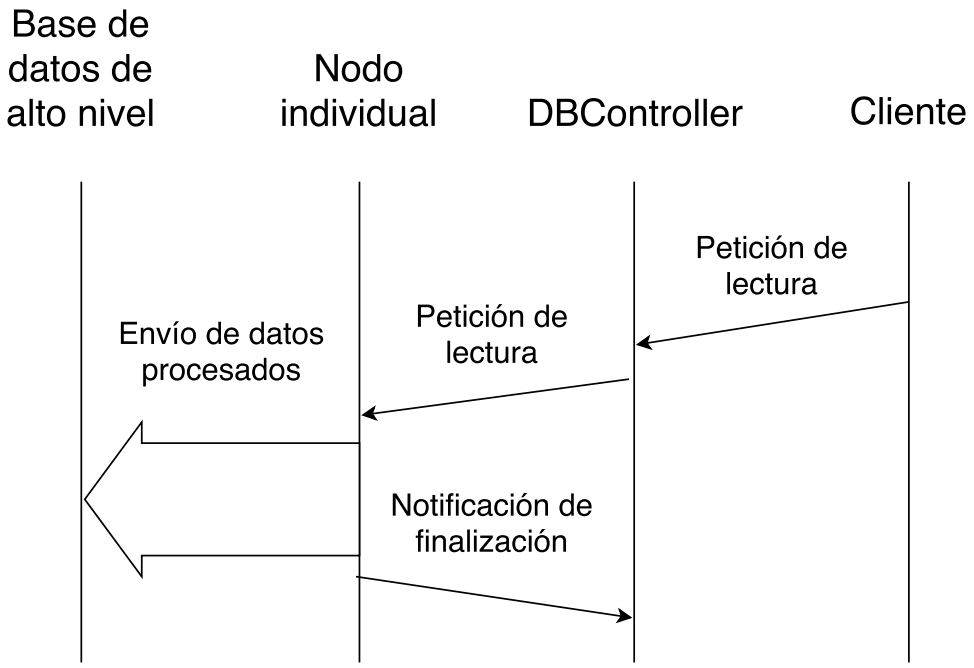

Figura 3.4: Proceso de lectura de la base de datos

almacenamiento, donde se almacenan en almacenamiento persistente al mismo tiempo que la componente de procesado los filtra, procesa y reenvía a la base de datos de alto nivel para realizar análisis más complejos.

4

Desarrollo

El desarrollo del sistema se dividió en varias fases que correspondían con los distintos elementos del sistema (**LogFeeder**, escritura a disco, lectura de disco y procesado).

4.1. LogFeeder

El diseño inicial de **LogFeeder** buscaba hacer el filtrado y procesado en el mismo nodo donde se recibían y almacenaban los logs. Como se explica en la sección 3.2.1, este diseño fue implementado y se pudo comprobar que, aunque era posible hacer un procesado muy eficiente, no se llegaba a cumplir el requisito de procesar al menos 3 millones de logs por segundo. Tampoco era posible hacer un filtrado lo suficientemente agresivo como para reducir los datos a una cantidad que pudiera ser almacenada en un único nodo. La solución a estos problemas consistió en cambiar el diseño para hacer un sistema distribuido. En este sistema tanto el procesado como el almacenamiento fueron divididos entre varios nodos. De este modo, el nodo de recepción (llamado **LogFeeder**) se encargaría únicamente de recibir, clasificar y distribuir los datos de forma uniforme entre los distintos nodos de almacenamiento.

Este componente se ha desarrollado en el lenguaje de programación C para obtener el máximo rendimiento. Los logs se reciben dentro de datagramas UDP en formato de **Syslog** para evitar los retardos causados por el control de congestión de TCP. El desarrollo inicial mostró que al usar directamente la API de sockets del sistema no se conseguía el rendimiento necesario para recibir 3 millones de logs por segundo. Esto se debe al sobrecoste introducido por la pila de red del sistema. Para arreglar este problema se decidió usar el driver **HPCAP** en vez del driver estándar de la tarjeta de red. Este driver, desarrollado en el laboratorio de investigación HPCN, permite recibir directamente los paquetes de red sin que pasen previamente por la pila de red del kernel.

Por otro lado, la emisión de los logs hacia las bases de datos de almacenamiento se hace a través de switches de alta velocidad dedicados únicamente al sistema, por lo que TCP es capaz de aprovechar el ancho de banda total del enlace. Para ello se copian logs a un buffer del mismo

tamaño del paquete que se enviará hasta que está lleno, momento en el que se envía a uno de los destinos siguiendo una estrategia de round robin. Al tener control sobre la red se pueden configurar las tarjetas y switches para usar *Jumbo Frames* permitiendo mandar paquetes de 9000 bytes en vez de los 1500 normales, reduciendo el sobrecoste introducido por las cabeceras de TCP.

El programa obtiene las IPs y puertos de las bases de datos de un fichero de configuración con el siguiente formato:

```
192.168.1.2 2000
192.168.1.3 2000
```

Si la conexión con uno de los nodos de base de datos se cierra, los logs se reparten uniformemente entre el resto de nodos hasta que se restablece la conexión cuando el nodo vuelve a estar disponible. De este modo no se pierden logs aunque fallen algunos nodos de base de datos.

Para saber cómo clasificar los logs el programa lee cuando se inicia dos ficheros de configuración. El primero indica las categorías de log que hay en cada combinación de IP y nivel de alerta de **Syslog**. Para hacer más legible este fichero la categoría puede tener un nombre. Sin embargo, al procesar los logs es más eficiente usar datos de tamaño fijo, por lo que el segundo fichero asocia cada nombre de categoría con una id numérica. Este fichero es usado más tarde por las componentes de lectura para volver a traducir del id al nombre. Si llega un log de una combinación de IP y nivel de alerta que no se encuentre en el primer fichero de configuración, el log es ignorado.

El formato del primer fichero de configuración es el siguiente:

```
 ip 0.0.0.1 {
    syslog 15 is apache
    syslog 16 is sendmail
}
```

Se puede comprobar que en esta IP hay dos servicios distintos asociados con dos niveles de alerta.

Por otro lado, el segundo fichero de configuración tiene el siguiente formato:

```
apache 1
sendmail 2
```

Esto permite que los mensajes internos del sistema usen la id numérica de 4 bytes especificada en el segundo campo para identificar cada categoría de log.

Dado que la clasificación de logs es la parte de **LogFeeder** que más recursos gasta, se decidió paralelizarla para aumentar el número de logs que se podían procesar por segundo. Para ello se usó el diseño que se muestra en la figura 4.1, donde un hilo productor y varios consumidores comparten datos mediante una cola circular de buffers sincronizada con *spinlocks*.

El productor copia los logs que recibe en el primer buffer vacío que encuentra en la cola circular. Cuando el buffer está lleno, modifica una variable para indicar a los hilos consumidores que puede ser leído e intenta acceder al siguiente buffer libre para continuar escribiendo. En paralelo a las escrituras del productor, los hilos consumidores toman buffers marcados como llenos para clasificar los logs que contienen y enviarlos a los nodos de la base de datos.

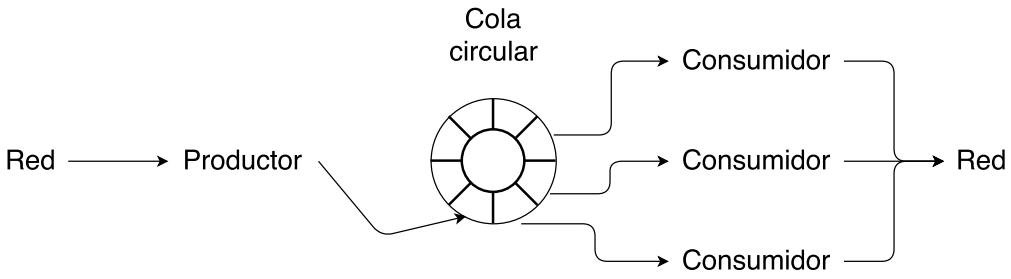

Figura 4.1: Diseño de LogFeeder

El acceso a la cola se hace a nivel de buffer para disminuir el número de veces que hay que bloquear y desbloquear los *spinlocks* ya que, a pesar de ser esperas activas, bloquear y desbloquear para cada log individual reduce mucho el rendimiento. Durante el desarrollo de **LogFeeder** se hicieron pruebas de rendimiento que permitieron determinar que el tamaño óptimo para estos buffers era de 300 MB.

4.2. Nodos de base de datos

La base de datos ha sido optimizada para usar RAIDs 0 de discos mecánicos. Es por esto que se almacenan los datos en ficheros de 1 GB. De este modo se puede aprovechar al máximo la velocidad de lectura y escritura secuencial de estos discos. También se guarda un índice de los ficheros en otro disco duro con el fin de que las búsquedas no afecten al rendimiento de escritura y lectura. El índice se guarda en una base de datos PostgreSQL, que permite usar el lenguaje SQL para realizar búsquedas de forma sencilla. El índice consiste en una tabla con los timestamps del primer y último paquetes almacenados en el fichero y el nombre del fichero.

4.2.1. Escritura

La componente de escritura recibe paquetes TCP enviados desde **LogFeeder** y copia los contenidos en un doble buffer de 1GB. Cuando un buffer se llena, se escribe a disco de forma asíncrona usando la interfaz AIO de POSIX y los nuevos datos pasan a escribirse en el otro buffer. Esta interfaz lanza un hilo cada vez que termina de escribir un fichero. Esto permite saber cuándo se puede volver a escribir en el buffer. Cada vez que se escribe un fichero a disco se introduce una entrada en la base de datos SQL indicando el nombre del fichero y los timestamps del primer y último paquete.

Al mismo tiempo que se introducen los logs en el buffer, un segundo hilo toma una muestra para procesarla y enviarla a la base de datos de alto nivel. A la hora de coger la muestra es importante no afectar a la escritura, por lo que este hilo solo procesa los logs que pueda copiar sin afectar al rendimiento. De este modo, si se reciben pocos logs se procesarán todos, mientras que si se reciben muchos solo se procesarán los que el sistema pueda copiar. El mecanismo de procesado será explicado en la sección 4.2.3.

4.2.2. Lectura

Para hacer lecturas se creó **DBController**, un programa que recibe peticiones de lectura de un cliente y las redirige a cada nodo de la base de datos. Estas peticiones se mandan a través de mensajes TCP donde se indican los timestamps de inicio y fin, la IP de la base de datos de alto nivel a la que se deben enviar los logs procesados, la proporción de logs a procesar y una lista de categorías a procesar. Para elegir la proporción de logs a procesar se usa un porcentaje y un tamaño de bloque. De este modo, si se pide un 10 % de los logs con tamaño de bloque 1 se procesarán 1 de cada 10 paquetes, mientras que si se usa tamaño de bloque 100 se procesarán 100 paquetes seguidos y se ignorarán los siguientes 900.

Además de **DBController**, se ha creado una API web que recibe órdenes en formato JSON. Esta API expone operaciones para crear nuevas peticiones de lectura, listar las lecturas en ejecución o encoladas y cancelar peticiones de lectura. Para facilitar la realización de pruebas durante el desarrollo también se implementó una interfaz web simple para exponer la API en un navegador. Esta interfaz no está diseñada para ser definitiva, sino que la componente de visualización será la encargada de añadir a su interfaz gráfica los formularios necesarios para hacer peticiones de lectura.

Para evitar bloquear al cliente HTTP durante toda la duración de la lectura, las peticiones se almacenan en una base de datos SQL. **DBController** monitoriza esta base de datos y redirige las peticiones de lectura o cancelación a cada nodo de la base de datos distribuida, manteniendo en todo momento actualizado el estado de cada petición en la base de datos SQL para que la API web pueda mostrar correctamente el estado del sistema.

DBController, la API web y la interfaz provisional han sido escritos en el lenguaje de programación Python y se comunican mediante una base de datos SQLite, ya que esta parte del sistema no necesita tanto rendimiento. Para implementar la API web y la interfaz se ha usado el *framework* **Flask** [25], que hace más fácil gestionar rutas HTTP.

En cada nodo de la base de datos hay un programa dedicado a hacer lecturas que usa dos hilos. El primero es el encargado de buscar en el índice los ficheros, leerlos, procesarlos y enviarlos mediante TCP a la base de datos de alto nivel. El segundo hilo gestiona la comunicación con el **DBController**, informándole cuando terminan las lecturas o esperando a nuevas peticiones de lectura o cancelación.

El proceso completo de lectura se muestra en la figura 3.4, mientras que el proceso para cancelar se muestra en la figura 4.2.

4.2.3. Procesado

Para hacer el procesado de los paquetes se barajaron distintas opciones que daban prioridad a distintos factores como la velocidad de proceso o la facilidad de uso.

El primer diseño fue el usado cuando el sistema estaba pensado para ser ejecutado en un único nodo. Consistía en crear un lenguaje dedicado exclusivamente al procesado de logs y diseñado para conseguir el mayor rendimiento posible. La versión básica que se implementó sólo conseguía procesar hasta 600000 logs por segundo usando todos los núcleos de un servidor, por lo que fue necesario cambiar el diseño para mover la componente de procesado a los nodos de almacenamiento y conseguir así mayor paralelismo. Otro problema de crear un lenguaje es que sería poco familiar para los usuarios del sistema final, ya que los administradores de sistema están

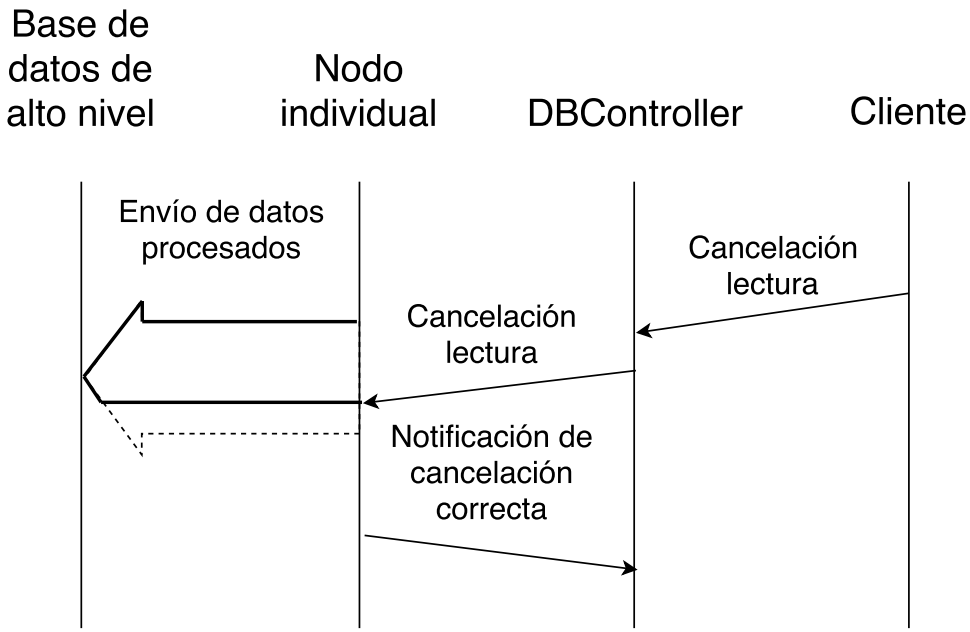

Figura 4.2: Proceso para cancelar una lectura

más acostumbrados a usar lenguajes de scripting como Python o AWK. Además, crear un nuevo lenguaje habría obligado a dedicar mucho tiempo de desarrollo para cada nueva funcionalidad deseada.

Es por estas razones que se barajó una segunda opción consistente en usar el lenguaje AWK y el uso de librerías que permiten llamarlo directamente desde código C. Esta opción también fue descartada dado que normalmente los campos dentro de las lineas de log están separados por varios símbolos distintos y el código AWK necesario para hacer el proceso de logs era poco legible.

La tercera opción fue la elegida finalmente y consiste en lanzar un proceso bash desde C. De este modo se pueden ejecutar scripts bash que usan pipelines de programas como sed y awk. Esta estrategia es la más familiar para los administradores de sistemas que gestionarían el sistema, ofreciendo también la ventaja de poder reutilizar programas ya existentes. Los logs se mandan al proceso bash a través de una tubería y los resultados se obtienen a través de otra. A cada categoría de log se le asocia un script de procesado en un fichero de configuración con el siguiente formato:

```
./processors/apache_processor.sh apache
./processors/sendmail_processor.sh sendmail
```

5 Resultados

Durante las pruebas se buscó comprobar que no se corrompieran los datos al pasar por el sistema y que el rendimiento cumpliera con los requisitos propuestos.

5.1. Pruebas de funcionamiento

Durante el desarrollo del sistema se realizaron las siguientes pruebas para comprobar la integridad de los datos

Se mantuvo un control del número de logs que entraban en cada etapa del sistema para comprobar que era el mismo que en la salida. En el caso del sistema entero la prueba consistió en contar el número de logs recibidos por LogFeeder y comprobar que era igual al número de logs obtenidos al hacer una lectura completa de la base de datos.

Una vez comprobado que el número de logs no variaba entre la entrada y la salida, se hicieron pruebas de cada sistema individual con el fin de verificar que no se duplicaran o borraran logs. En el caso de LogFeeder se comprobó que los hilos consumidores leían correctamente de la cola circular sin saltar o leer dos veces el mismo buffer y que, una vez leído el buffer, los datos se mandaran en el mismo orden por la red. Por otro lado, se comprobó que los logs recibidos en la base de datos fueran los mismos que los obtenidos al leer de ella. Entre estas dos etapas se produce un desorden de los logs debido al uso de varios hilos. Sin embargo, el uso del protocolo TCP garantiza que no se habrán perdido o duplicado datos.

Estas pruebas se realizaron tanto en uno como en varios nodos usando distinto número de hilos y de nodos de base de datos para comprobar que las distintas combinaciones dieran siempre resultados correctos.

Para probar las lecturas de la base de datos se usó la interfaz web, creando y cancelando peticiones de lectura en distintas fases para comprobar que el comportamiento fuera siempre correcto.

5.2. Pruebas de rendimiento

A continuación se muestra el rendimiento obtenido por el sistema. Primero se dan los resultados obtenidos al probar cada elemento del sistema de forma individual y después los resultados de probar el sistema entero.

5.2.1. Entorno de pruebas

Para realizar las pruebas de rendimiento se usaron tres servidores distintos. Uno de ellos (NRG) contaba con un RAID y los otros dos (neon y skadi) con discos individuales. El primer servidor estaba conectado directamente a los otros dos mediante interfaces de 10 Gbps, por lo que se usó para ejecutar LogFeeder en todas las pruebas.

5.2.2. LogFeeder

Datos en memoria

En estas pruebas se comprobó el rendimiento de LogFeeder enviando datos cargados previamente en memoria RAM y variando el número de hilos consumidores para comprobar que se cumplieran los requisitos de rendimiento y determinar el número óptimo de hilos. También se varió el tamaño de los logs para obtener una visión más completa de cómo se comporta el sistema. Estas pruebas permiten conocer el rendimiento máximo del sistema, ya que no se encuentra limitado por la velocidad máxima de 10 Gbps que daría una tarjeta de red.

Las figuras 5.1 y 5.2 muestran los resultados de estas pruebas. Se puede comprobar que un único hilo es capaz de procesar los 3 millones de logs por segundo si el tamaño del log es menor de 291 bytes, valor que se encuentra por encima del tamaño medio de los logs. En la figura 5.2 también se puede ver que, si se usan al menos dos consumidores, en todos los casos se consiguen velocidades de proceso mayores a los 10 Gbps y la tarjeta de red será siempre el cuello de botella del sistema. Por último, estas pruebas muestran que usar más de 3 consumidores no mejora el rendimiento de forma sustancial e incluso puede perjudicarlo en algunos casos.

También es importante comprobar que el rendimiento del sistema es constante, ya que de lo contrario se podrían perder logs. Esto se comprobó ejecutando LogFeeder durante más de una hora y midiendo el número de logs de 291 bytes procesados cada 100 milisegundos. En la figura 5.3 se pueden ver los resultados obtenidos de la ejecución completa con 4 consumidores, mientras que la tabla 5.1 muestra las estadísticas obtenidas al analizar el sistema estabilizado con distinto número de consumidores. Se puede comprobar que el rendimiento llega a ser constante varios segundos después de iniciar las pruebas. Durante estos segundos el sistema está en la fase de inicialización, con los buffers siendo reservados y llenados por primera vez.

HPCAP

Para completar las pruebas de logfeeder, se generó una traza con logs de formato syslog de tamaño mínimo (caso peor) y se enviaron mediante un generador de tráfico hardware a NRG a través de una interfaz de 10 Gbps. Se comprobó que no se perdían paquetes y que se saturaba el enlace, lo que demostró que el cuello de botella era la tarjeta de red.

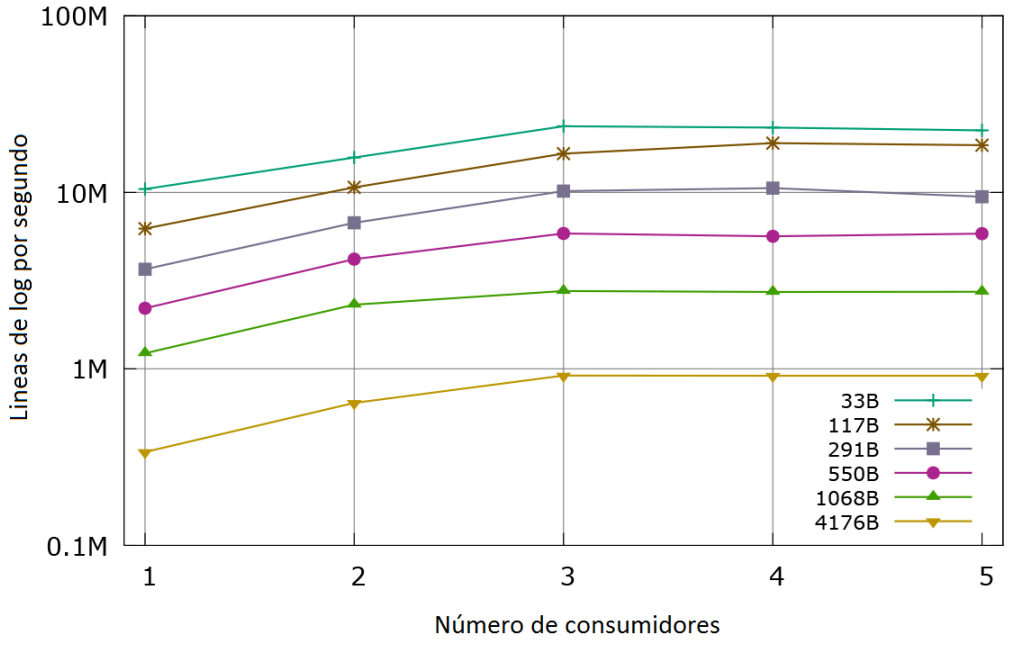

Figura 5.1: Logs por segundo en función del tamaño de log

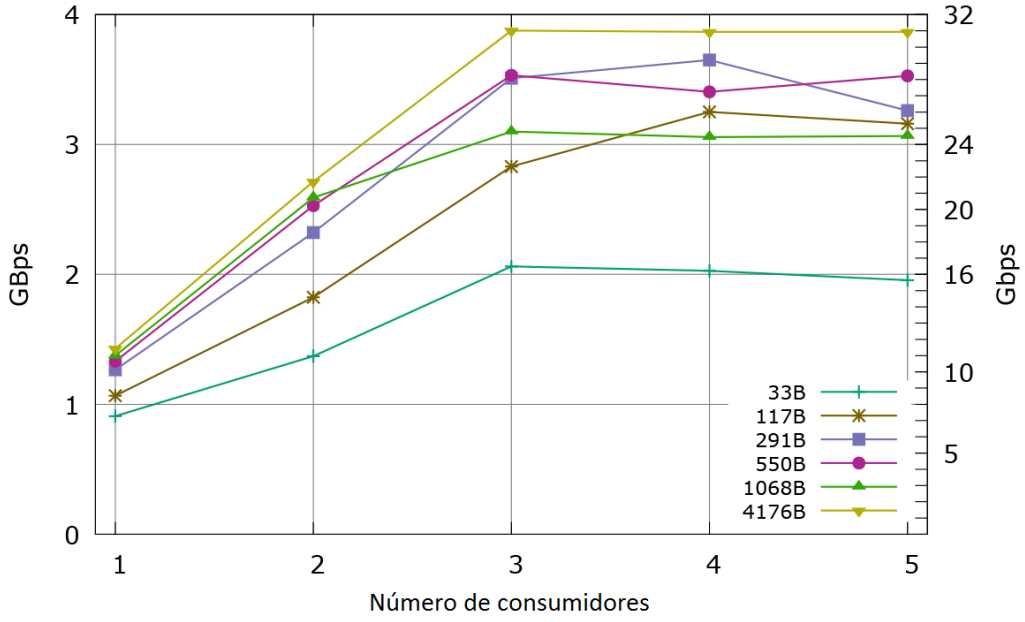

Figura 5.2: Gbytes y Gbps en función del tamaño de log

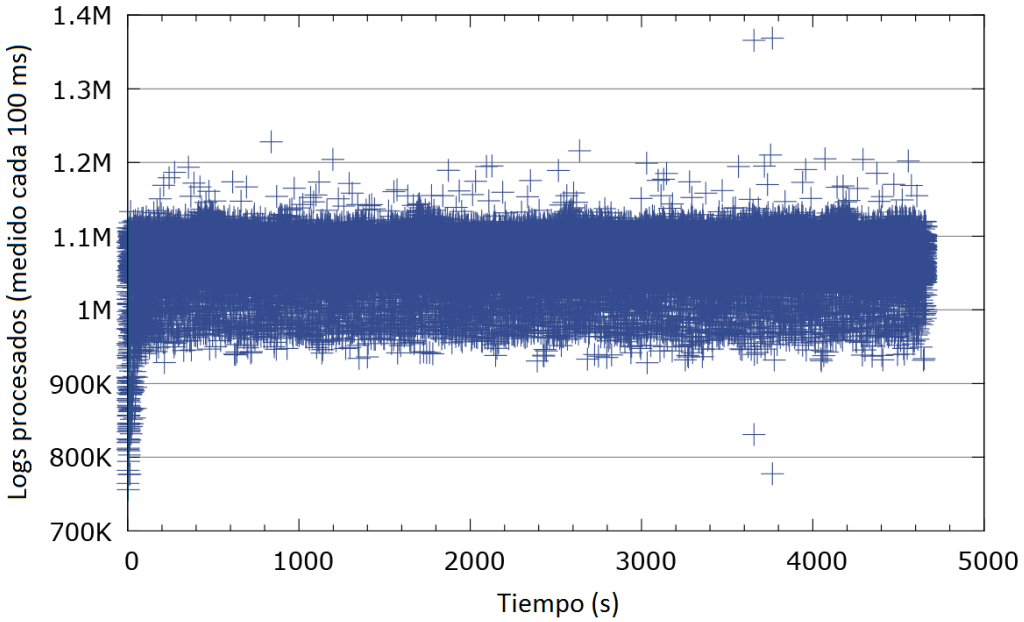

Figura 5.3: Rendimiento usando 4 consumidores y logs de 291 bytes

Tabla 5.1: Número de logs de 291 bytes procesados cada 100 ms.

Número de consumidores	Media	Mediana	Desviación estándar
1	512600	516100	17945.03
2	693200	701900	25254
3	1038000	1044000	17588.47
4	1070000	1073000	30756.99

5.2.3. Base de datos

Las pruebas de rendimiento de la base de datos se centraron en comprobar el efecto que las lecturas tenían sobre las escrituras, dado que estas últimas deben tener prioridad para no perder logs.

Para ello se escribió un programa que simulaba las escrituras a disco que hace la base de datos. En este programa se podía configurar la tasa de escritura. Al mismo tiempo que se ejecutaba este programa con una tasa determinada, se lanzaba otro que leía del disco a la máxima velocidad posible y se medían las velocidades de escritura y lectura tras dejar que se estabilizaran. De este modo se puede concluir que si la tasa de escritura medida es distinta a la configurada, las lecturas están perjudicando al rendimiento de escritura.

Tanto en las pruebas con estos programas como al ejecutar el sistema se usó la utilidad *ionice* de Linux para dar la máxima prioridad al programa de escritura y la mínima al de lectura.

Las pruebas se realizaron en discos individuales capaces de obtener una velocidad de escritura secuencial de hasta 160 MB/s. Los resultados de la figura 5.4 muestran que si se limita la tasa de escritura al 50 % (80 MB/s) hacer lecturas paralelas no tiene efecto. Sin embargo, si se supera esta tasa las lecturas hacen que empeore el rendimiento. Gracias a esta prueba se pudo determinar

que para que el sistema de almacenamiento no sufra pérdidas de rendimiento es necesario tener suficientes discos para que en ningún momento se supere el 50 % de tasa de escritura.

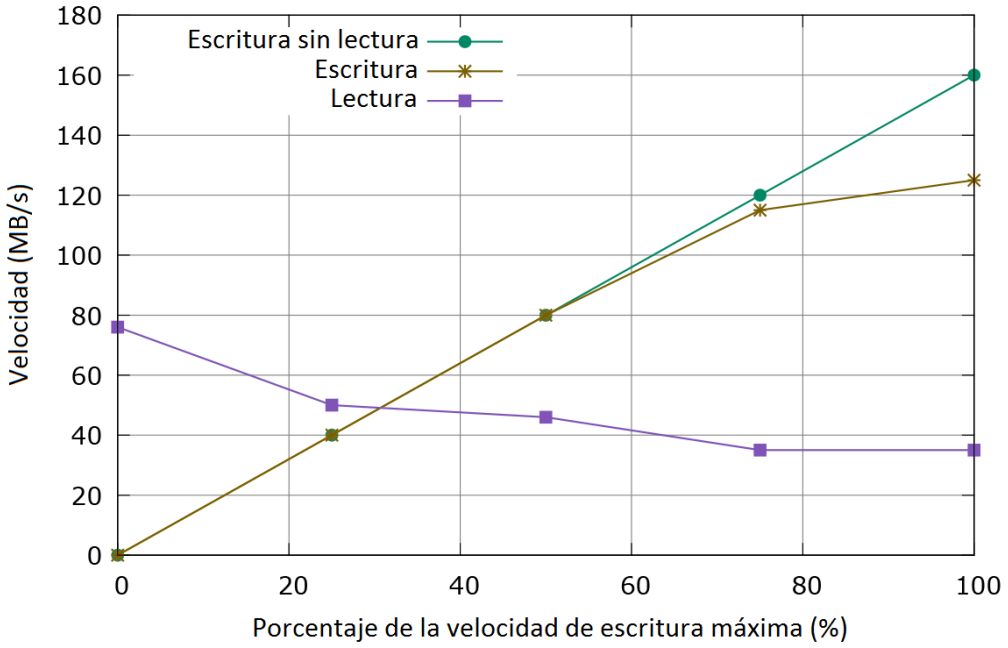

Figura 5.4: Efecto de las lecturas sobre la tasa de escritura

Estos resultados fueron confirmados en pruebas del sistema completo, en las que se limitó la velocidad de LogFeeder para terminar con un 50 % de tasa de escritura y se pudo comprobar que las lecturas no causaban pérdidas.

5.2.4. Procesado

Usar pipes para mandar los logs a un proceso bash y leer los resultados limita la velocidad de procesado que se puede conseguir. Para determinar la velocidad máxima se hicieron lecturas de disco y se procesaron con un script que escribía los mismos datos que leía. De este modo se comprobó que se podían procesar hasta 370k logs de tamaño medio por segundo. En un sistema ejecutándose en producción esta velocidad se multiplica por el número de nodos de almacenamiento.

5.2.5. Sistema completo

En las pruebas del sistema entero se buscó comprobar que todos los elementos del sistema funcionaran adecuadamente sin que ninguno introdujera un cuello de botella. Para ello se enviaron a LogFeeder los 3 millones de logs por segundo que se propusieron como objetivo y, tras almacenar varios minutos de logs, se realizaron lecturas simultáneas.

Estas pruebas se hicieron en dos escenarios distintos:

Un nodo

Primero se ejecutó el sistema entero en un único nodo (NRG). Este nodo cuenta con un RAID 0 de 10 discos, por lo que es capaz de almacenar 3 millones de logs de tamaño medio sin problemas. Se comprobó que, al estar usando menos del 50 % de la tasa de escritura del RAID, las lecturas no afectaban al rendimiento del sistema.

Varios nodos

La segunda prueba del sistema completo buscaba comprobar que se pudieran usar varios nodos interconectados sin perder rendimiento. Para ello se ejecutó LogFeeder en NRG y los nodos de base de datos en neon y skadi. Dado que cada uno de estos dos ordenadores tiene 7 discos individuales, se ejecutaron el mismo número de nodos de base de datos para aprovecharlos. Las pruebas demostraron que el sistema también cumplía con los requisitos de rendimiento al usar varios nodos, incluyendo lectura y procesado.

6

Conclusiones

En este trabajo se ha desarrollado un sistema de almacenamiento y procesado de logs capaz de tratar con más de 3 millones de logs de tamaño medio por segundo.

Este sistema permite mantener un registro completo de todos los logs generados por un gran número de servidores. Estos logs pueden ser utilizados para estudiar incidencias y determinar cómo prevenirlas en el futuro. El sistema supera el rendimiento de los que existían previamente usando una fracción del número de servidores, lo que permite monitorizar de manera económica un gran número de dispositivos generadores de logs.

Aunque la mejora de rendimiento se consigue a cambio de cierta pérdida de flexibilidad, el sistema puede tomar una muestra de los logs y enviarlos a algún sistema de alto nivel en el que hacer un procesado en tiempo real que permita detectar incidencias inmediatamente. De este modo se puede aprovechar la flexibilidad de los sistemas existentes sin que sea necesario usar un gran número de nodos.

Otra ventaja del sistema es que no es necesario instalar software específico en los generadores de logs, sino que se pueden utilizar herramientas estándar como **Syslog**.

Además de conseguir un escalado vertical gracias al uso de lenguajes nativos, el sistema puede escalar horizontalmente gracias a que los componentes que lo forman son independientes unos de otros. Esto permite conseguir velocidades incluso mayores a las propuestas en los requisitos iniciales, haciendo que el sistema pueda seguir siendo usando aunque en el futuro aumente el número de logs a almacenar y procesar.

Como trabajo futuro, se modificará el modo en que **LogFeeder** determina las categorías de los logs, ya que sobreescribir el nivel de alerta hace que se pierda información que puede ser útil. En su lugar se determinará la categoría a partir de la IP de origen y el puerto de destino. También se estudiará la posibilidad de reordenar los logs por timestamp una vez lleguen a cada nodo de la base de datos, ya que el diseño actual puede introducir algo de desorden debido al uso de varios hilos emisores en **LogFeeder**.

Bibliografía

[1] BalaBit. How to configure syslog-ng pe to cooperate with splunk. https://www.balabit.com/documents/pdf/syslog-ng-pe-whitepaper-splunk.pdf, 2015. [Online; accedido 29-March-2016].

[2] R. Bodkin. Mapr releases commercial distributions based on hadoop. http://www.infoq.com/news/2011/07/mapr, 2011. [Online; accedido 29-March-2016].

[3] S. Corp. On-disk microbenchmark. http://symas.com/mdb/ondisk/, 2014. [Online; accedido 29-March-2016].

[4] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008. [Online; accesed 29-March-2016].

[5] FluentD. Fluentd architecture. http://www.fluentd.org/architecture, 2015. [Online; accedido 29-March-2016].

[6] C. Kalantzis. Revisiting 1 million writes per second. http://techblog.netflix.com/2014/07/revisiting-1-million-writes-per-second.html, 2014. [Online; accedido 29-March-2016].

[7] J. Kepner, W. Arcand, D. Bestor, B. Bergeron, C. Byun, V. Gadepally, M. Hubbell, P. Michaleas, J. Mullen, A. Prout, et al. Achieving 100,000,000 database inserts per second using accumulo and d4m. In *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*, pages 1–6. IEEE, 2014.

[8] J. Kreps. Benchmarking apache kafka: 2 million writes per second (on three cheap machines). https://engineering.linkedin.com/kafka/benchmarking%2dapache%2dkafka%2d2%2dmillion%2dwrites%2dsecond%2dthree%2dcheap%2dmachines, 2014. [Online; accedido 29-March-2016].

[9] N. Marz and J. Warren. *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co., 2015.

[10] E. POINT. Benchmarking top nosql databases. http://www.datastax.com/wp-content/themes/datastax-2014-08/files/NoSQL_Benchmarks_EndPoint.pdf, 2015. [Online; accedido 29-March-2016].

[11] M. Richards. *Software architecture patterns*. O'Reilly, 2015.

[12] B. SA. Sending messages directly to elasticsearch. https://www.balabit.com/sites/default/files/documents/syslog-ng-ose-latest-guides/en/syslog-ng-ose-guide-admin/html/configuring-destinations-elasticsearch.html, 2015. [Online; accedido 29-March-2016].

[13] J. Scott. Loading a time series database at 100 million points per second. https://www.mapr.com/blog/loading-time-series-database-100-million-points-second, 2014. [Online; accedido 29-March-2016].

[14] Scylla. Scylla architecture. http://www.scylladb.com/technology/architecture/, 2015. [Online; accedido 29-March-2016].

[15] Scylla. Scylla vs. cassandra benchmark. http://www.scylladb.com/technology/scylla-vs-cassandra-benchmark/, 2015. [Online; accedido 29-March-2016].

[16] Scylla. Scylla vs. cassandra benchmark (cluster). http://www.scylladb.com/technology/scylla-vs-cassandra-benchmark-cluster/, 2015. [Online; accedido 29-March-2016].

[17] H. Shreedharan. *Using Flume: Flexible, Scalable, and Reliable Data Streaming.* .º'Reilly Media, Inc.", 2014.

[18] E. Silva. Fluentd: a high performance unified logging layer. https://www.linux.com/news/enterprise/high-performance/147-high-performance/847237-fluentd-a-high-performance-unified-logging-layer, 2015. [Online; accedido 29-March-2016].

[19] E. Silva. Unifying events & logs into the cloud. http://events.linuxfoundation.org/sites/events/files/slides/unifying_events.pdf, 2015. [Online; accedido 29-March-2016].

[20] Splunk. Splunkit v2.0.2 results and ec2 storage comparisons. http://blogs.splunk.com/2013/06/06/splunkit-v2-0-2-results-ec2-storage-comparisons/, 2013. [Online; accedido 29-March-2016].

[21] Splunk. Splunk sizing and performance: Doing more with more. http://blogs.splunk.com/2014/05/07/splunk-sizing-and-performance-doing-more-with-more/, 2014. [Online; accedido 29-March-2016].

[22] Elasticsearch. Elasticsearch nightly benchmarks https://benchmarks.elastic.co/index.html, 2016 [Online; accedido 16-June-2016]

[23] V. VMoreno, P. Santiago del Rio, J. Ramos, D. Muelas, J. Garcia-Dorado, G.-A. F.J., and J. Aracil. Multi-granular, multi-purpose and multi-Gb/s monitoring on off-the-shelf systems. http://onlinelibrary.wiley.com/doi/10.1002/nem.1861/abstract, 2014. [Online; accedido 29-March-2016].

[24] Yang, Fangjin and Tschetter, Eric and Léauté, Xavier and Ray, Nelson and Merlino, Gian and Ganguli, Deep. Druid: a real-time analytical data store. Proceedings of the 2014 ACM SIGMOD international conference on Management of data

[25] Flask. http://flask.pocoo.org/, [Online; accedido 7-Septiembre-2016].